Combining Static and Dynamic Validation of MPI Collective Communication

Emmanuelle Saillard, Patrick Carribault, Denis Barthou

Combining Static and Dynamic Validation of MPI Collective Communications

Emmanuelle Saillard
CEA, DAM, DIF
F-91297 Arpajon, France
emmanuelle.saillard@cea.fr

Patrick Carribault
CEA, DAM, DIF
F-91297 Arpajon, France
patrick.carribault@cea.fr

Denis Barthou
University of Bordeaux / INRIA
Bordeaux, France
denis.barthou@labri.fr

ABSTRACT

Collective MPI communications have to be executed in the same order by all processes in their communicator and the same number of times, otherwise a deadlock occurs. As soon as the control-flow involving these collective operations becomes more complex, in particular including conditionals on process ranks, ensuring the correction of such code is error-prone. We propose in this paper a static analysis to detect when such situation occurs, combined with a code transformation that prevents from deadlocking. We show on several benchmarks the small impact on performance and the ease of integration of our techniques in the development process.

General Terms

Verification

Keywords

MPI, debugging, collective, static analysis, correctness

1. INTRODUCTION

Most of scientific applications in High-Performance Computing rely on the MPI parallel programming model to efficiently exploit a supercomputer and reach high parallel performance. Based on the distributed-memory paradigm, this model exposes two ways to express communications between tasks/processes: point-to-point and collective. While point-to-point functions involve only two tasks, collective communications require that all processes in a communicator invoke the same operation. Each process does not have to statically invoke such collective function at the same line of the source code, but the sequences of collective calls in all MPI processes must be the same and corresponding function calls should have a compatible set of arguments. Due to the control flow inside a MPI program, processes may execute different execution paths. Such behavior may cause errors and deadlocks difficult for the user to detect and analyze.

To tackle this issue, this paper presents a two-step analysis to detect incorrect collective patterns in SPMD MPI programs. For each function, we first identify at compile-time the code fragments calling collectives that may deadlock and the control-flow parts that may lead to such situation. Warnings are issued during this phase. Then, we transform the identified code fragments in order to dynamically capture these situations before they arise. The runtime overhead of this instrumentation is limited since only the pieces of code calling collectives that can deadlock are modified. In case of actual misuse, the application stops with an explicit error message highlighting both the collectives and the control-flow code responsible for this situation.

EuroMPI'13 Madrid, SPAIN

1.1 Motivating example

The following simple example illustrates the potential issues with collective communications.

```
void f( int r ) {
    if( r == 0 ) MPI_Barrier(MPI_COMM_WORLD);
    return;
}
void g( int r ) {
    f(r);
    MPI_Barrier(MPI_COMM_WORLD);
    exit(0);
}
```

Assume here that `g` is called by all processes. Depending on the value of the input parameter `r`, a process will execute or not the barrier in the `if` statement in `f`. If `r` is not uniformly true or false among all MPI processes, some tasks will be blocked in `f` while the remaining process ranks will reach the barrier in `g`. These processes will then terminate, while the first ones will be in a deadlock situation at the barrier in `g`. The machine state when the deadlock occurs does not help to identify the cause of the deadlock.

We propose in this paper methods (i) to identify the conditional in `f` as the cause for a possible deadlock, using compiler analysis and (ii) to prevent from deadlocking using a code transformation. As the value of `r` is unknown at compile time and might be the same for every MPI process, the dynamic state of control flow has to be checked in order to prevent from entering a deadlock state. Transforming the previous example would lead to the code:

```
void f( int r ){
    MPI_Comm c; int n1, n2;
    if( r == 0 ) {
       MPI_Comm_split(MPI_COMM_WORLD, 1, 0, &c);
       MPI_Comm_size( c, &n1 );
       MPI_Comm_size( MPI_COMM_WORLD, &n2 );
       if ( n1 != n2 ) MPI_Abort();
       MPI_Comm_free(&c);
       MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Comm_split(MPI_COMM_WORLD, 0, 0, &c);
    MPI_Comm_size( c, &n1 );
    MPI_Comm_size( MPI_COMM_WORLD, &n2 );
    if ( n1 != n2 ) MPI_Abort();
    MPI_Comm_free(&c);
```

```
    return;
}
```

In order to partition processes according to their behavior regarding the conditional, two calls to the collective `MPI_Comm_split` are inserted in the code: one before the barrier operation with color 1 (2^{nd} parameter of the call), and one before the `return` statement with color 0. All processes call the `MPI_Comm_split` collective, whatever their execution path. However, both the communicator for color 1 and the one used in the following barrier should have the same size, otherwise the function is incorrect and an `MPI_Abort` is issued in order to prevent from deadlocking.

1.2 Context and Contributions

This paper focuses on scientific SPMD applications parallelized with MPI. We suppose functions calling collective operations are called by all processes in the same order and the same number of time. A function is said to be a *correct function* regarding blocking collective communication if all MPI processes entering the function eventually exit without leaving any process blocked inside a collective operation. We consider that mismatching collectives are the only source of deadlock, neglecting other possible sources such as infinite loops, blocking IOs and other deadlocks which would require dedicated analysis. While all types of collectives are handled, collective operations are assumed to be called on the same communicators, with compatible arguments. In this context, this article makes the following contributions:

- Identification of collective callsites that may lead to deadlocks, and of control-flow codes responsible for such situation within each function. If all execution paths can be executed in parallel, the identified collective corresponds to deadlock situations (no over-approximation then).
- Limited instrumentation based on the previous analysis to prevent collective errors at execution time, pinpointing the control-flow divergence responsible for such errors.
- Full implementation inside a production compiler, experimental results on MPI benchmarks and applications.

1.3 Outline

Section 2 describes related work on MPI debugging and analysis, focusing on collective operations. The static analysis detecting collective issues is presented in Section 3, and the code transformation to capture incorrect functions is described Section 4. Section 5 shows experimental results.

2. RELATED WORK

Related work on MPI code verification can be organized in 3 categories: (i) static analyses, (ii) online dynamic analyses and (ii) trace-based dynamic analyses.

Static tools.

Few MPI validation tools rely exclusively on static analysis. This class of tools mainly based on model checking does require symbolic program execution, at the expense of combinatorial number of schedules or reachable states to consider, making this approach challenging. TASS[9], a successor of MPI-SPIN follows this approach: using model checking and symbolic execution, it checks numerous program properties explicitly annotated with pragmas. If a property is violated (as wrong order of collective calls) by exploring reachable states of the model built, an explicit counter-example is returned to the user in the form of a step-by-step trace through the program showing the values of variables at each state of the synthesised model. Unlike TASS, our static check analyzer requires no source-code modifications to verify collective matching since it is integrated inside a compiler. Potential errors are automatically returned to the programmer with their context (including the line of the erroneous conditional) through a low-complexity control-flow graph analysis. A pragma-based approach however could be useful to improve our static analysis (for example by tagging MPI rank dependent variables), thus reducing false-positive possibilities. Besides, in our approach, the combinatorial aspect of detecting effective mismatch is avoided by the runtime checking.

Online dynamic tools.

Dealing with dynamic tools able to check collective operations, we can mention DAMPI[13], Marmot[6, 8], Umpire[12, 8], MPI-CHECK[7, 8], Intel Message Checker (IMC)[2, 8] and MUST[5, 4]. Umpire, Marmot and MUST rely on a dynamic analysis of MPI calls instrumented through the MPI profiling interface (PMPI). They are able to detect mismatching collectives either with a timeout approach (DAMPI, Marmot, IMC and MPI-Check) or with a scheduling validation (Umpire and MUST). Methods performing deadlock detections through a timeout approach are known to produce false positives, for example in case of abnormal latencies. In our approach we detect possibly erroneous calls at compilation time (filtering phase) but by analyzing the code function by function, our method detects errors sooner compared to most other tools. DAMPI uses a scalable algorithm based on Lamport Clocks (vector clocks focused on call order) to capture possible non deterministic matches. For each MPI collective operation, participating processes update their clock, based on operation semantics. Umpire, limited to shared memory platforms, relies on dependency graphs with additional arcs for collective operations to detect deadlocks, whereas in Marmot, an additional MPI process performs a global analysis of function calls and communication patterns. Both of these approaches, however, might be limited by scalability. MUST overcomes the limitations of Umpire and Marmot in term of scalability by relying on a tree-based layout. Another tool, MPI-CHECK instruments the source code at compile time replacing all MPI calls with modified calls with extra arguments. In our approach, we perform a runtime check, taking advantage of the compile time analysis results (code locus and potential error filtering) in order to scale to large programs, avoiding instrumentation of the whole MPI interface or systematic code instrumentation like in MPI-CHECK.

Validation can also be done inside MPI libraries such as in an extension of MPICH, allowing collective verification for the full MPI-2 standard[3, 10]. The detection of runtime deadlock causes is however limited to the information available to the MPI routines. Comparatively to most dynamic analysis tools, our method provides more precise errors including the conditionals responsible with the help of our static check.

Trace-based dynamic tools.

IMC (recently replaced by the Intel Trace Analyzer, an online analysis) uses a different approach as it collects all MPI-related information in trace files. The post-mortem analysis of these traces tends to be difficult and with limited scalability due to the trace sizes, correlated to the number of cores.

Our detection of incorrect functions combines both static and dynamic approaches. The compilation analysis finds all locations that may cause potential errors and the code is transformed in order to stop the execution whenever a function is incorrect. More precisely, when a deadlock situation occurs in a run, an error message is returned with both the location and the type of the collective and the control-flow code responsible for this situation. The program then

aborts allowing a program state exploration with a debugger. As our dynamic check is performed by a lightweight library (see Algorithm 2), it is also independent from the MPI implementation.

3. COMPILE-TIME VERIFICATION

The first step consists in a static analysis of the *control-flow graph (CFG)* for each function. The CFG is defined as a directed graph (V,E) where V represents the set of basic blocks and E is the set of edges. Each edge $u \longrightarrow v \in E$ depicts a potential flow of control from node u to v. A node $u \in V$ has a set of successors denoted as $SUCC(u)$. Moreover, we assume that the underlying compiler appends two unique artificial nodes for entry and exit points.

3.1 Algorithm Description

Based on this structure, Algorithm 1 details the steps of our static compile-time analysis to detect if a function is correct (see Section 1.2). The algorithm takes as input the CFG of the current function and outputs nodes that may lead to collective errors (from S) and a set O of collectives that may deadlock. This set will be given as parameter to the instrumentation detailed in Section 4.

The principle of the algorithm is the following: we compute for each node of the CFG the number of collectives executed to reach the node from the function entry. This number will be 0 for nodes before the first collective (including the node with the first collective), 1 for nodes reached after one collective and so on. When multiple paths exists, nodes can have multiple numbers, at most the number of collectives in the function. Loop backedges are removed to have a finite numbering and the algorithm is applied to the CFG of each loop separately. For nodes with collectives, these numbers correspond to the possible *execution ranks* of the collective within the function. In a correct function, for any given rank k, all execution paths from entry to exit should traverse the nodes of rank k with the same collective operation. A function is not correct if there are nodes with out-going paths through nodes of execution rank k and other paths that do not traverse nodes of rank k or with different collectives. These nodes correspond to possible control-flow divergence leading to deadlocks, since it is possible to execute a different number of collectives, or in a different order. They can be computed using the iterated postdominance frontier [1].

A node u postdominates a node v if all paths from v to exit go through u. We extend this relation to sets: a set U postdominates a node v if all paths from v to exit go through at least one node of U. The postdominance frontier of a node u, $PDF(u)$ is the set of all nodes v such that u postdominates a successor of v but does not strictly postdominate v. If $\gg$ denotes the postdominance relation,

$$PDF(u) = \{v \mid \exists\, w \in SUCC(v), u \gg w \text{ and } u \not\gg v\}$$

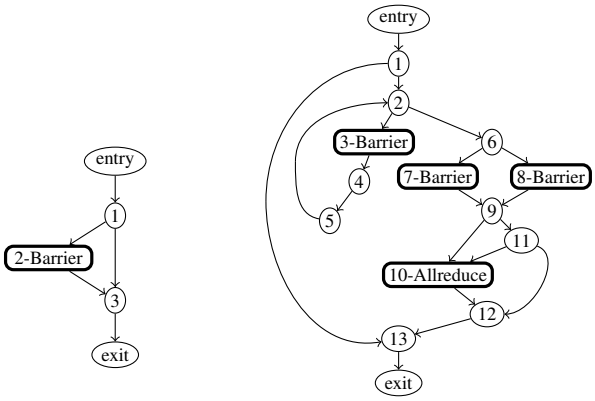
This notion is extended to a set of nodes U. The iterated postdominance frontier PDF^+ is defined as the transitive closure of PDF, when considered as a relation [1].

Algorithm 1 describes this computation applied to each function and loop, entry and exit being defined then as loop entry and exit. Execution rank computation corresponds to a simple traversal of the acyclic CFG, counting traversed nodes with collectives. Then for each execution rank r, the nodes calling the same function c, at rank r are clustered into $C_{r,c}$. The iterated postdominance frontier of this set corresponds to nodes that can lead both to the execution of such collective or not. Note that the algorithm can handle any collective operation, but since comunicators are expected to be the same along collectives, and other parameters are assumed to be correct, only the name of the collective is used in the algorithm.

3.2 Example

Algorithm 1 Step 1 - Static Pass

```
1: function STATIC_PASS(G = (V,E))                    ▷ G: CFG
2:     O ← ∅
3:     S ← ∅                                          ▷ Output set
4:     Remove loop backedges in G to compute execution ranks for nodes
       with collectives
5:     for r in node ranks do
6:         for c in collective names of execution rank r do
7:             C_{r,c} ← {u ∈ V | u of rank r, of collective name c}
8:             S ← S ∪ PDF^+(C_{r,c})
9:             O ← O ∪ c
10:        end for
11:    end for
12:    Output nodes in S as warnings, nodes in O for Step 2.
13: end function
```



(a) CFG of function f (b) CFG from a Benchmark

Figure 1: Examples of Control Flow Graphs

Figure 1(a) depicts the CFG extracted from the initial code of Section 1.1. It contains 3 nodes: the first one represents the `if` statement while the second one contains the `if` body with the collective call. Finally the last one denotes the `return` instruction. The algorithm considers the set $C_{0,\texttt{Barrier}} = \{2\}$ corresponding to the collective `Barrier`. As its iterated postdominance frontier is node 1, the algorithm finally outputs a warning for the condition located in node 1 and flags the collective `Barrier` for the following dynamic analysis (set O).

Figure 1(b) presents another CFG extracted from a real benchmark. This example contains 2 collectives: `Barrier` (nodes 3, 7 and 8) and `Allreduce` (node 10). The algorithm first removes the backedge $5 \rightarrow 2$ from the loop and computes ranks. Nodes $7,8$ are of rank 0, 10 of rank 1. For the collectives in $C_{0,\texttt{Barrier}} = \{7,8\}$, the iterated postdominance frontier corresponds to node 1. Note that node 6 is postdominated by $\{7,8\}$ according to the definition of previous section. $C_{1,\texttt{Allreduce}}$ contains only node 10 and $PDF^+(C_{1,\texttt{Allreduce}}) = \{1,9,10\}$. Indeed from these nodes, it is possible to execute the `Allreduce` or not. Finally, the same algorithm is applied once more on the graph with nodes $\{2,3,4,5\}$ corresponding to the loop, without the backedge. Node 2 is marked as entry and exit. This node is the only one in the iterated postdominance frontier of the barrier in node 3. To sum up, node 1 decide of the number of execution of `Barriers` in $7,8$, nodes $9,11$ decide of the number of execution of `Allreduce` and node 2 is responsible for the number of `Barriers` executed in node 3.

3.3 Algorithm Proof

Previous algorithm computes the set S of control-flow nodes that

have execution paths with different number or type of collectives, from the node to the exit node. We prove that the algorithm finds a non-empty set O if and only if the function is incorrect, and the nodes in S correspond exactly to the nodes that lead to a deadlock.

Consider a node u from S, computed from the function CFG. u belongs to a set $PDF^+(C_{r,c})$ for some rank r and collective function c. It implies that there is an outgoing path from u that goes through the collective c of rank r, and another path that reaches the exit node without going through a collective c of same rank. If the second path never reaches a collective c (any rank) and if both paths are executed by different tasks, then some tasks will wait at the collective c while the other tasks will either wait at another collective (a deadlock) or exit the function (incorrect function). In both cases, function is incorrect. If both paths traverse the same collective c, since the ranks are different, one of the paths has more collectives c than the other. Again, this leads to an incorrect function.

The algorithm is applied on each loop separately. This separate analysis identifies at least loop exit nodes as control-flow nodes that may be responsible for deadlocks, when the loop calls collectives. Indeed, static analysis does not count iterations and collectives in loops may be executed a different number of times for each process.

Now consider an incorrect function: when executing this function with multiple tasks, some task may reach the exit node while other tasks are waiting at a collective c inside the function. This collective is executed in a node u with rank r. u belongs to $C_{r,c}$. If the iterated postdominance frontier $PDF^+(C_{r,c})$ is empty, it would imply that all nodes of the function are postdominated by a collective c at rank r. This means that all tasks would execute the same k^{th} collective c, whatever the path taken. As this is not the case, $PDF^+(C_{r,c})$ is not empty, $C_{r,c} \subseteq O$.

4. EXECUTION-TIME VERIFICATION

The code fragments leading potentially to incorrect functions, detected with the previous analysis, are transformed in order to raise an error message at the execution time: whenever MPI tasks take execution paths that cannot lead to the same number of collectives, in the same order, the program stops. This section presents the code transformation involved.

4.1 Algorithm Description

We introduce the function `CC` depicted in Algorithm 2 to check if a future call to a collective operation will ultimately generate an error. It takes as input the communicator com_c related to the collective call c and a color i_c specific to the type of collective.

Algorithm 2 Library Function To Check Collectives (CC)

```
1: function CC(com_c, i_c)
2:     MPI_Comm c'
3:     MPI_COMM_SPLIT(com_c, i_c, 0, &c')
4:     MPI_COMM_SIZE(com_c, &n)
5:     MPI_COMM_SIZE(c', &n')
6:     if n ≠ n' then
7:         MPI_ABORT()
8:     end if
9:     MPI_COMM_FREE(&c')
10: end function
```

Relying on the `CC` function, Algorithm 3 describes the instrumentation for the execution-time verification. It takes as parameter the function CFG and the set O generated by Algorithm 1. For each node n containing a call to the collective c, a new communicator grouping processes traversing n is created just before calling c. A piece of code testing the size of the resulting communicator and the original communicator used for the collective is then added. If both sizes are different, an error is issued and the program is aborted through a call to `MPI_Abort`. This process is repeated for each collective operation c in set O. Finally, in the closest node of collective nodes that post-dominates and joins all paths of the CFG, `MPI_Comm_split` with the color 0 is added to eventually catch up processes not calling any additional collective. Figure 2 presents the transformation achieved by Algorithm 3 on the CFG Figure 1(b).

Algorithm 3 Step 2 - Selective Instrumentation

```
1: function INSTRUMENTATION(G, O)
2:                                  ▷ G: CFG, O: set created by Algorithm 1
3:     for c ∈ O do
4:         for n in nodes containing a call to collective c do
5:             x ← com_{n,c}
6:             Insert call to CC(x, i_c) before the call to c
7:         end for
8:     end for
9:     Insert call to CC(x, 0) before the return statement
10: end function
```

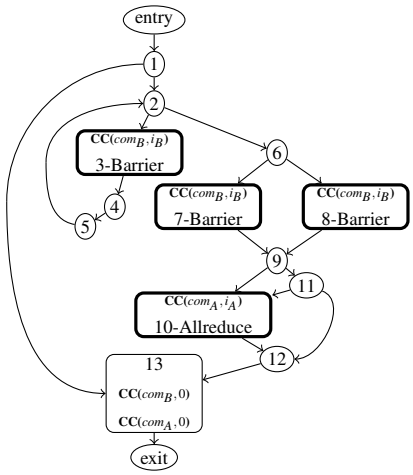


Figure 2: Instrumented CFG Figure 1(b) (Algorithm 3)

4.2 Correctness Proof

Algorithm 3 is correct if all deadlocks situations are captured by the instrumentation and if the new collectives do not generate a deadlock themselves.

We define a *control sequence* the sequence of collective calls executed by a process in a program execution. For an execution of a given function, a control sequence is denoted as $c_1 c_2 .. c_n$ with c_i the i-th collective called. Algorithm 3 rewrites each collective c_j from the set O into $s_j c_j$ corresponding to the function `MPI_Comm_split` called by CC based on the color j and the initial collective c_j. The function `MPI_Comm_split` with color 0 denoted as s_0 is added after all collective nodes. To ease the proof, we will assume that this conditional rewriting, performed only for collectives found by the static analysis, is conducted for all collectives of the control sequence. Consequently, a sequence $c_1 .. c_n$ becomes $s_1 c_1 .. s_n c_n s_0$.

If all control sequences are the same for all processes, the function executes with no deadlock. By applying Algorithm 3, the modified control sequence are still identical, this algorithm does not introduce deadlocks.

If a function deadlocks due to collective operations,

- Either a process calls a collective communication c_i while another process calls a collective function c_k with $k \neq i$. The

control sequence of both processes differ only with their last collective, c_k and c_i, and both are prefixed by $c_1..c_{i-1}$.

- Or a process calls a collective communication while another one exits the function (a deadlock may occur at a later point in the execution or outside of the function). The control sequence of the process exiting the function is $c_1..c_{i-1}$ and the process inside the function executes the same prefix sequence with one more collective c_i.

In the first case, the algorithm changes both control sequences into $s_1c_1..s_{i-1}c_{i-1}s_i$ and $s_1c_1..s_{i-1}c_{i-1}s_k$. These sequences stop with s_i and s_k since CC(x,i) and CC(x,k) lead to an error detection and abort. Hence the modified function no longer deadlocks.

In the second case, the algorithm changes both control sequences into $s_1c_1..s_{i-1}c_{i-1}s_i$ for the process inside the function, and $s_1c_1..s_{i-1}c_{i-1}s_0$ for the one trying to leave the function. Note that the process is stopped before leaving the function since CC(x,i) and CC(x,0) both abort, generating an error message. Again, the modified function does not deadlock any more. To conclude, Algorithm 3 is indeed correct and prevents all deadlock situations.

5. EXPERIMENTAL RESULTS

We implemented our analysis in a GCC 4.7.0 plugin as a new pass inserted inside the compiler pass manager after generating the CFG information. Thus, this solution is language independent, allowing to check MPI applications written in C, C++ or FORTRAN. The pass applies Algorithms 1 and 3. The application needs to be linked to our dynamic library for runtime checking (see Algorithm 2). Our static analysis is simple to deploy in existing environment as it does not modify the whole compilation chain.

This section presents experimental results obtained on a representative C++ MPI application, EulerMHD[14], solving the Euler and ideal magnetohydrodynamics equations both at high order on a 2D Cartesian mesh, and on the MPI NAS Parallel benchmarks (NASPB v3.2) using class C [11]. We selected six benchmarks from the NASPB to have both C and Fortran programs.

5.1 Test Platform and Methodology

All experiments were conducted on Tera 100, a Petaflopic supercomputer with an aggregate peak performance of 1.2 PetaFlops. It hosts 4,370 nodes for a total of 140,000 cores. Each node gathers four eight-core Nehalem EX processors at 2.27 GHz and 64 GB of RAM. All performance results are computed as the average over 8 runs (compilation or execution) with BullxMPI 1.1.14.3.

5.2 Static Check Results

At compile time, a warning is returned to the programmer when a potential deadlock situation is detected. The following example shows what a user can read on `stderr` for NAS benchmark IS:

```
is.c:In function 'main':
is.c:1093:1: warning: STATIC-CHECK: MPI_Reduce may
not be called by all processes in the communicator
because of the conditional line 923
Check inserted before MPI_Reduce line 994
```

This warning provides the name of the collective that may deadlock (`MPI_Reduce`) and the line of the conditional leading to the collective call (line 923). This collective call is instrumented at line 994 as described in Algorithm 3. Notice that the line number where the control flow divergence may lead to a deadlock is not close to the collective call. However, this case corresponds to a false positive result because the conditional statement is a test over the number of processes (exit if more than 512 MPI processes).

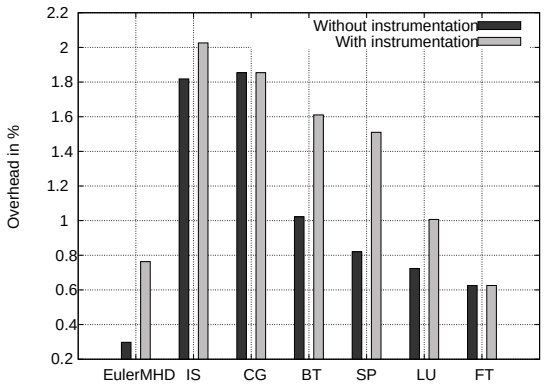


Figure 3: Overhead of average compilation time with and without code instrumentation

Figure 3 details the overhead of compilation time when activating our GCC plugin. This overhead remains acceptable as it does not exceed 3%. It is presented with and without the code generation which accounts for the insertion of *CC* function calls (see Algorithm 2). This specific step is mainly responsible for the overhead except for CG and FT. Indeed, according to the static analysis, these benchmarks are correct, no collective operation is instrumented.

Table 1: Compilation and Execution Results

Benchmark	# collective calls	# nodes in S	% instrumented collectives	# calls to CC
EulerMHD	28	14	36%	26
BT	9	5	78%	8
LU	14	2	14%	6
SP	8	5	75%	7
IS	5	2	40%	3
CG	2	0	0%	0
FT	8	0	0%	0

For each benchmark, Table 1 presents the number of nodes found by the Algorithm 1 (set S) and the number of static calls to a collective communication. This table shows that the static analysis is able to reduce the amount of instrumentation needed to check the collective patterns (third column). Reducing further the number of instrumented collectives would require an inter-procedural data-flow analysis on the nodes in S. For all these nodes, the control-flow does not depend on process ranks and the functions are correct. Such analysis is outside the scope of this paper and is left for future work. Finally, the last column depicts the number of calls to the CC function during the execution of the benchmarks.

5.3 Execution Results

Figure 4 shows the overhead obtained for NASPB class C from 4 to 512 cores (CG and FT have no overhead as no collective is instrumented). The overhead does not exceed 18% and tends to slightly increase with the number of cores. Figure 5 presents weak-scaling results for EulerMHD from 1 to 1,280 cores where the overhead remains comparable with the same increasing trend. Table 1 presents the number of runtime checks. Processes about to call collectives identified as potential deadlock sources are counted. If some processes are missing, the abort function is called to stop the program before deadlocking. An error is printed to `stderr` with the line number, the collective name and conditionals responsible:

```
DYNAMIC-CHECK: Error detected on rank 0
DYNAMIC-CHECK: Abort before MPI_Barrier line 47
DYNAMIC-CHECK: See warnings about conditional(s)
               line 45
```

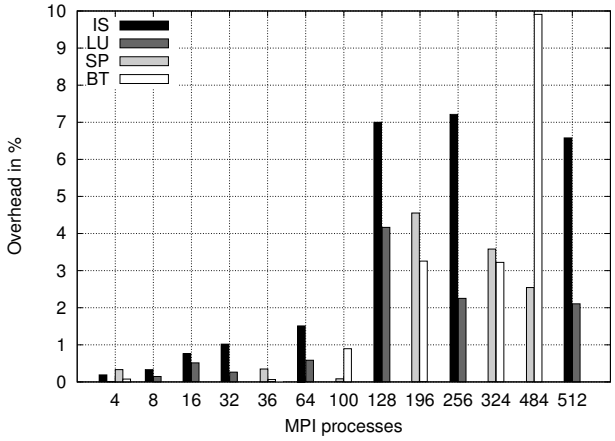


Figure 4: Execution-Time Overhead for NASPB (Class C, Strong Scaling)

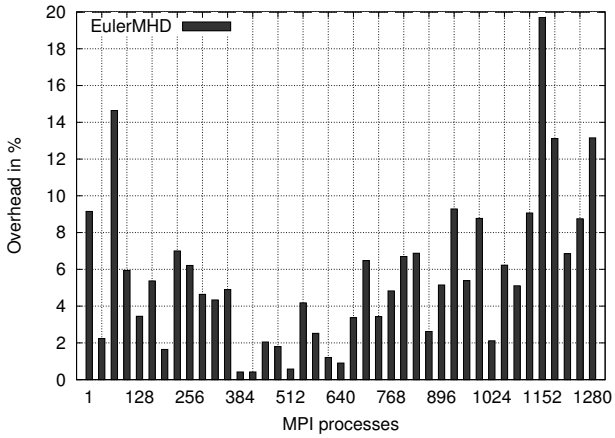


Figure 5: Execution-Time Overhead for EulerMHD (Weak Scaling)

6. CONCLUSION AND FUTURE WORK

In this paper we described a two-phase analysis to detect collective patterns in MPI programs that can cause deadlocks. The first pass statically identifies the reduced set of collective communications that may eventually lead to potential deadlock situations, and issues warnings. Using this analysis, a selective instrumentation of the code is achieved, displaying an error, synchronously interrupting all processes, if the schedule leads to a deadlock situation. If all execution paths can be executed in parallel, we have shown that the number of collectives transformed is minimal. This method is easily integrated in GCC compiler as a plugin, avoiding recompilation. We have shown that the compilation overhead is very low (3%). Dealing with the runtime overhead, it could become non-negligible at larger scale as our analysis adds collectives for instrumentation. However, with the help of collective selection, the runtime overhead remains acceptable (less than 20%) at a representative scale on a C++ application.

Although it satisfies both scalability and functional requirements, our analysis is only intra-procedural with the possible drawback of missing conditional statements out of function boundaries. Moreover, our analysis is focused on a particular error and should be extended to cover common verification, for example, MPI call arguments, such as different communicators. These improvements are currently under development, and the analysis is being extended to inter-procedural analysis, gathering more data-flow information at compile-time in order to further reduce the number of instrumented collectives. Furthermore, our approach is a preliminary work setting the basis for a wider set of analysis combining static and dynamic aspects and extended to OpenMP and hybrid (OpenMP + MPI) parallelisms.

7. ACKNOWLEDGMENTS

This work is (integrated and) supported by the PERFCLOUD project. A French FSN (Fond pour la Société Numérique) cooperative project that associates academics and industrials partners in order to design then provide building blocks for a new generation of HPC datacenters.

8. REFERENCES

[1] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM TOPLAS*, pages 13(4):451–490, 1991.

[2] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *SE-HPCS*, pages 78–82, 2005.

[3] C. Falzone, A. Chan, E. Lusk, and W. Gropp. Collective error detection for MPI collective operations. In *PVM/MPI*, pages 138–147, 2005.

[4] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *Supercomputing*, pages 30:1–30:11, 2012.

[5] T. Hilbrich, M. Schulz, B. de Supinski, and M. Muller. MUST: A scalable approach to runtime error detection in MPI programs. Parallel Tools Workshop, 2010.

[6] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI analysis and checking tool. In *PARCO*, pages 493–500, 2003.

[7] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. Concurrency and Computation: Practice and Experience, pages 15:93–100, 2003.

[8] S. Sharma, G. Gopalakrishnan, and R. M. Kirby. A survey of MPI related debuggers and tools. 2007.

[9] S. Siegel and T. Zirkel. Automatic formal verification of MPI based parallel programs. In *PPoPP*, pages 309–310, 2011.

[10] J. L. Träff and J. Worringen. Verifying collective MPI calls. In *PVM/MPI*, pages 18–27, 2004.

[11] NASPB site:http://www.nas.nasa.gov/software/NPBl.

[12] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, pages 51–51. ACM/IEEE, 2000.

[13] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *Supercomputing*, pages 1–10, 2010.

[14] M. Wolff, S. Jaouen, and H. Jourden. Hight-order dimensionally split lagrange-remap schemes for ideal magnetohydrodynamics. In *Discrete and Continuous Dynamical Systems Series S*. NMCF, 2009.